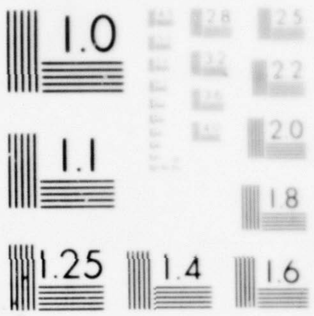
AD-A036 296 ROYAL SIGNALS AND RADAR ESTABLISHMENT CHRISTCHURCH (E--ETC F/G 9/2
REDUNDANCY AND RECOVERY IN THE HIVE VIRTUAL MACHINE.(U)
MAY 76 J M TAYLOR
UNCLASSIFIED RSRE-76010 DRIC-BR-54783 NL

1 OF 1
AD
A036296

END
DATE
FILMED
3-77



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

UNLIMITED

BR54783

Report No.76010

Report No.76010

ADA036296

0

R.S.

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
CHRISTCHURCH.

Redundancy and recovery in the HIVE virtual machine

by

J.M. Taylor

D D C
[illegible]
C

COPYRIGHT ©

CONTROLLER HMSO LONDON

1976

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Christchurch, [illegible]

[illegible]

MAY 1976

UNLIMITED

Procurement Executive, Ministry of Defence
Royal Signals and Radar Establishment, Christchurch

D D C
RECEIVED
FEB 14 1977
C

14 RSRE-
Report No. 76010

6

REDUNDANCY AND RECOVERY IN THE HIVE VIRTUAL MACHINE

by

10 J. M. TAYLOR

12 30p.

11 May 76

18 DRIC

19 BR-54783

SUMMARY

The HIVE project is concerned with studying high level virtual machine architectures suitable for designing and implementing large, high-integrity transaction processing applications such as communication switching and database access systems. The main aim of the work is to develop a unified set of structural concepts and components in terms of which all the different and often conflicting design aspects of such systems can be coherently expressed. The approach followed has been to embody these ideas in the specification of a high integrity virtual machine, HIVE, and to implement HIVE and evaluate it experimentally by using it to implement test-bed applications systems.

This work has involved considering not only the architecture of the overall run-time software during normal operation, but also various selective approaches to data protection, redundancy and recovery, and methods for easing the task of describing, generating and integrating all the software of an application in an incremental, evolutionary way.

This report is concerned primarily with the first two of these areas. It presents the main points of the HIVE system as it stands at present, with particular emphasis on the selective data redundancy and recovery aspects, and also discusses the rationale behind the main design principles of the system.

A

ACCESSION for
NTIS White Section ✓
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

May 1976

4B
409 807

CONTENTS Page

1. INTRODUCTION

The HIVE project at RSRE (formerly SRDE) is concerned with investigating high level virtual machine architectures suitable for implementing large, dedicated transaction processing systems such as communications switching and database access systems. Currently, such systems are often designed and implemented according to many different and frequently conflicting structural principles. These in turn are often derived either from various aspects of the basic system requirement (eg high integrity operation, modifiability, size, real-time response etc) or else they result from basing the design on particular existing hardware and software components (eg a particular computer system, operating system or language).

A basic aim of the HIVE research has been to develop a more unified conceptual framework for designing and implementing these types of system, providing a set of structual concepts and components in terms of which all the various aspects of a system can be expressed coherently. To allow experimental study and evaluation of these ideas, we have proceeded by designing and implementing a high integrity virtual machine, HIVE, and using it to implement, initially, the application software for a test-bed message switching system.

The development of HIVE itself has involved 3 major areas:-

- the architecture of the overall run-time software systems in normal operation;
- methods for allowing high integrity operation by selectively protecting critical data during normal operation and reconstructing the system after a fault;
- methods for easing the task of generating and integrating all the software for a given application in an incremental way.

The present report is concerned mainly with the first two areas. Its aim is to present the main points of the HIVE system as it stands at the moment, and in particular to describe our current approach to data protection and recovery. However, we begin with a discussion of some of the various types of structure found in transaction processing systems, in order to introduce and explain most of the main design principles on which HIVE itself is based.

2. TRANSACTIONS AND FUNCTIONS

The first obvious components of a transaction processing system are the actual transactions being processed. Typically, such a system processes many randomly arising transactions apparently independently and in parallel, and each transaction will be represented during its lifetime in the system by one or more associated data areas. The actions needed to process any given transaction can be represented as a set of operations or functions which are to be executed on the data of that transaction. Different transactions may need different sets of functions, and functions may be hierarchically related, sequentially related or both.

Many different criteria may be used to partition the overall processing required into discrete functions, each corresponding to a different structural view of the system. For example, functions may be defined so as to give well-defined, compact interfaces between them for which correctness and validity checks can be specified, or to give clearly separated programming tasks during implementation. Functions may be defined that can be asynchronously scheduled for execution at run time (eg to meet real time requirements, one function might need to run 100 times more frequently than another), or chosen to facilitate modification and expansion after cut-over. Alternatively, the system might be hierarchically organised with functions defined to support various different levels of abstraction. The system might be designed around its database organisation with operations grouped together according to which files they access, or more general considerations of protection and redundancy could be the dominant theme, with functional boundaries defined so they can be enforced by hardware protection facilities and to allow redundant copies of data areas etc to be maintained securely at run time.

It is clear that these criteria may conflict if considered independently and so some common framework in which they can all be expressed coherently is desirable. We must therefore look at some of the underlying structural components they imply.

3. CODE AND DATA - VERSIONS

The operation of a function on a transaction is likely to be implemented as the execution of some module(s) of program code on the data area(s) representing the transaction. Two other kinds of data area are also likely to be required, namely workspace areas providing temporary, scratchpad workspace private to the functional program itself, and one or more database areas. These exist permanently in the system and will be read and/or updated by the functional code in the course of its execution for any particular transaction. There may be many of these database areas in the whole system, any of which may be accessed by more than one function and together they constitute the system's dynamic, long-term memory. Examples include routing tables, user database files, transaction audit files etc, etc.

These various code and data modules must be stored in physical hardware. They must be loaded into fast core store or its equivalent when execution is required and we assume that in large, high integrity systems they must also be held on some form of backing store such as discs both for economic reasons (too expensive to hold all the code and data in core) and also to permit recovery from faults, as we shall discuss later. For our present purposes, we will refer to this backing storage as discs although its actual nature is not important. This immediately implies that a given code or data object may exist in several different forms or versions simultaneously within the system, for example an executable version in core, working version on disc A, backup version on disc B, cold start versions on discs A and B and perhaps also some source code versions.

Our basic structual components "transactions" and "functions" (however defined) are thus likely to be formed out of more primitive components such as code and data modules, each of which in turn may be regarded as a composite object consisting of several coexisting versions.

4. PARALLEL PROCESSING - CO-OPERATING SINGLE THREAD PROGRAMS

Existing high level programming languages are almost entirely designed around producing single thread programs, ie programs able to contain only a single thread or path of execution at run time. Such languages are thus often well suited to producing the code for a single function processing a single transaction. However, an overall transaction processing system is likely to consist of a whole set of sequentially and hierarchically related functions which must be able to deal with many transactions concurrently in parallel at various arbitrary stages in their progress through the system. This raises several important points about structure and languages.

4.1 Separate Single Thread Programs

If we are producing a large, complex system, we should not aim to produce all the software as a single, large program because this will make management of development and modification very difficult. Rather, the software should be implemented as a set of separately compilable programs or program modules, each corresponding to a separate function or group of functions in the system. If we wish to use current high level languages, and all the experience they represent, in writing these programs, then each functional program should be written in single thread form, to process one transaction at a time.

4.2 Co-operating Programs - Parallel Processing

The actual run-time system will thus consist of an interconnected network of separate functional programs, and this overall system cannot be single thread even if it contains only a single hardware processor. This is because it must maintain the appearance of processing many transactions in parallel and so at any instant some or many of the separate programs will be at various arbitrary stages in processing the various transactions currently in the system, even though their execution has been temporarily suspended because no processor is available to run them. This simultaneous existence of a number of separate threads of execution at run time is usually called parallel processing and the separate threads are usually defined as or associated with system components called processes. If, as is usually the case, there are fewer real processors than processes wanting to run, multiprogramming is required to time-share them.

4.3 Process Strategies

Each time the single thread program for a given function is applied to process the data of a given transaction, this is regarded as a new, separate execution of this program. The mechanism by which such a program is applied to deal with a number of transactions, either physically sequentially with a single incarnation of that program, or physically in parallel with multiple incarnations, is in general beyond the scope of single thread programming languages. The facilities required are in some ways analogous to those of a job control language but serious problems of integrity and protection are also involved. This inherent need for parallelism plus the desirability of using single thread programming languages for writing function modules, however they are defined, are thus fundamental structural considerations. They have lead directly to two of the basic elements of the HIVE system design, namely:-

- a virtual machine architecture (see below) which provides a set of asynchronous, independent virtual processors, each able to execute a functional process, together with means for these to communicate;
- a system building facility based on a system description language, SYDEL, which allows separate functional programs written in single thread form using a conventional high level language (Coral 66 at present) to be configured as a set of co-operating, asynchronous processes mapped onto an interconnected network of HIVE virtual processors.

5. ABSTRACTION AND VIRTUAL MACHINES

We mentioned in passing that functions may be hierarchically related, ie that one may call on another, that they may be implemented in terms of simpler components such as code and data modules, and that similarly a transaction, as it progresses through the system, may be represented as an evolving set of associated data modules. These are examples of the process of abstraction which is fundamental to structuring complex systems in comprehensible ways. It involves analysing a given component in terms of simpler, more "primitive", "lower-level" components (the top-down approach); or synthesising a "higher level" component as suitable combinations of existing, already defined components (the bottom-up approach). Its aim of course is to restrict the complexity of any one level (ie the number of distinct components and their interactions) to comprehensible proportions, at the expense of loss of detail. Abstraction is thus only possible in well-structured systems, ie those in which the loss of detail at a given level is unimportant because it does not affect the essential behaviour being represented at that level.

Abstraction is a very general concept. One way of making it more specific in order to apply it systematically to dedicated computer systems is to define the levels of abstraction required as abstract or virtual (computing) machines. More specifically, the virtual machine at the i'th level of abstraction, VM(i), may be defined as the complete set of facilities and resources needed for specifying and executing level i programs. The facilities of VM(i) thus include an instruction set in terms of which level i programs may be expressed and including in particular means for manipulating the (abstract) resources of VM(i) (see below) and means for ensuring data protection, redundancy and recovery if required. Facilities must also be provided for invoking any compilation and system building processes needed to convert source-form programs into an executable component of a run-time system. The resources of VM(i) include storage resources for holding the various versions of program and data; input-output resources via which programs can communicate with each other and the outside world; and processor resources which can be applied to execute programs in storage on data in storage according to certain conventions.

Our approach in designing the HIVE system is based on trying to identify and specify a particular set of facilities and resources as

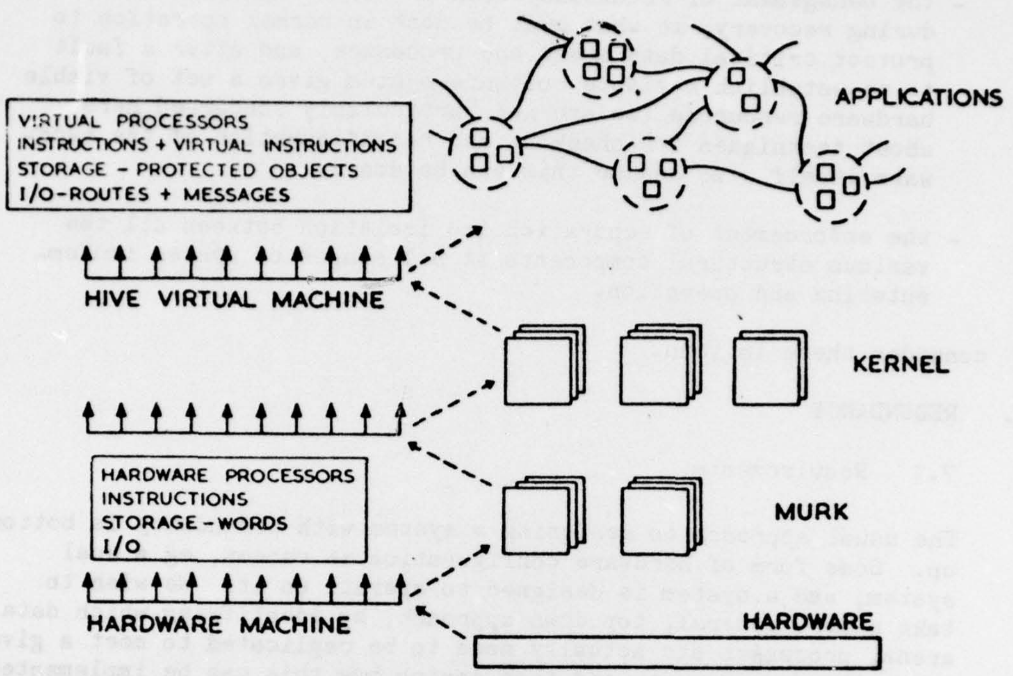


Fig 1 HIVE Virtual Machines - Abstractions & Implementations

a stable, standardised high level virtual machine. (Fig 1) This should be, on the one hand, at a sufficiently high level (with respect to current hardware machines) to provide useful simplifications, abstractions and hardware independence for the applications programmer (ie the programmer writing for this virtual machine and above). On the other hand, it should not be at so high a level that it is only suitable for one specific application.

Before discussing virtual machines any further, we must first consider one other important influence on the way an overall system may be structured, namely the requirement for high integrity operation and the associated need for protection, redundancy and recovery facilities.

6. HIGH INTEGRITY OPERATION

We define high integrity operation as the ability to maintain correct transaction processing through a high proportion of fault incidents of all kinds. The problems of achieving this fall into two main categories. The first is what is done before a fault, during normal operation, to prevent and eliminate faults (eg by enforcement and checking of structure), to detect faults, and to protect critical data areas etc against the effect of faults using redundancy techniques. The second category is the response to faults when they occur and this includes many stages such as localisation, isolation, fallback, initial recovery, diagnosis, repair, restoration and final recovery.

Our intention in this paper is to concentrate on the two key ways in which high integrity requirements affect the system structure and the virtual machine design. These are:-

- the management of redundancy both in normal operation and during recovery, ie what must be done in normal operation to protect critical data areas and processes, and after a fault to re-establish a viable software system given a set of viable hardware resources (we are not particularly concerned here about techniques for checkout and reconfiguration of the hardware itself - we assume this can be done).

- the enforcement of separation and isolation between all the various structural components at all stages of system implementation and operation.

W consider these in turn.

7. REDUNDANCY

7.1 Requirements

The usual approach to designing a system with redundancy is bottom up. Some form of hardware configuration is chosen, eg a dual system, and a system is designed to operate on it. We wish to take a more general, top down approach, by identifying which data areas, processes etc actually need to be replicated to meet a given integrity requirement, and then seeing how this can be implemented efficiently. The practicability of this approach depends on certain characteristics of dedicated, transaction processing systems. In particular, only part of the software of such systems is intimately linked to real-time events in the outside world, in the sense that if a fault occurred the effect would be immediately apparent as loss of unrepeatable input data or loss of output. If we wish to avoid this, we have no option but to physically replicate these critical processes and data on physically separate hardware, and we do not consider this further in this paper.

The remainder of the software (which in many cases is the bulk of it) is more decoupled from external, real-time events, so that if a fault occurs, there is some time available in which some form of <u>reprocessing</u> can be attempted before the external behaviour of the system becomes unacceptable. In other words, instead of providing complete physical replication of all processes and data in these "less-real-time" parts of the system, we have the possibility of replicating processes in time, ie running them again in some way if a fault occurs, and only providing actual physical replication of those code and data areas which are essential to allow processing to be retried.

From the hardware point of view, then, we wish to assume a more general configuration in which the amount of redundancy is not fixed to start with and to which redundancy can be added selectively in various ways and in relatively small increments, eg some form of store-based multiprocessor system consisting of a number of processors, core store modules, disc stores and input-output modules. The aim is to provide a given level of integrity with less redundancy than would be needed in a dual or triplicated configuration, and possibly with better multiple fault tolerance and better flexibility and expandability.

Let us now consider what forms of recovery and reprocessing might be required and how they might be achieved. We have established so far that the system is likely to be implemented as a set of single thread, functional programs executing as a network of parallel processes (NB: there is not necessarily a one-to-one correspondence between programs and processes, eg a given program might be associated with more than 1 process). In turn, a given process carrying out a given function for a particular transaction is likely to be implemented as a number of store areas in core and on discs, holding one or more versions of various types of data including code, database data, transaction data and workspace data.

We can thus give a general statement of the recovery and processing requirement using an 'outside in' or 'worst first' approach from two viewpoints, as follows.

1. It must be possible to recover from major crashes such as loss of core or a faulty processor provided a viable hardware configuration still exists, and this is likely to involve major recovery and reconstruction action eg complete reloading of core and garbage collection of discs. It is desirable to recover from less major faults such as loss of an individual disc unit or corruption of the data of a particular transaction with less drastic recovery action, eg reloading a disc, reprocessing particular transactions or repeating certain processes, without disturbing the rest of the system.

2. It must be possible to re-establish a 'cold start' or 'day one' version of the entire system, with no memory of any of its previous activity, provided a viable hardware configuration still exists. It is highly desirable to be able to recover the database data areas comprising the system's long-term memory in a form from which processing of new and possibly also existing transactions can be resumed. It is also desirable to be able to recover the data of transactions currently in the system in a form from which their processing can be completed or repeated.

We consider these from the 'outside in' again, starting with the recovery of the complete system after a major crash, since it turns out that the more minor or local recovery actions can then be dealt with by quite straightforward extensions to the basic approach. These will be discussed later in the description of HIVE itself.

7.2 Cold Start Recovery

The reconstitution of a 'cold-start' version of the complete system can be provided quite readily by requiring that:-

1. All the code of the system is made read-only so that it is never modified by normal operation of the run-time system. Particular instances or versions of various code areas may of course be corrupted by a fault at runtime or replaced by other versions in the course of modification procedures. We ignore the latter for our present purposes although it can be included by obvious extensions.

2. A set of similarly read-only constant data areas is provided which contains presets or initialisations for all

the data areas in the system which are read-write at run time.

3. Enough copies of these code and constant data areas are held on separate storage media eg discs, to give an adequate probability that an uncorrupted copy can be found and loaded after a crash by a recovery bootstrap.

This replicated read-only code and constant data thus forms an ultimate <u>backstop</u> from which the system can resume processing with no memory of any of its previous activity, ie containing no <u>dynamic</u> data generated by any previous execution of the system.

7.3 Dynamic Recovery

The real nub of the redundancy and recovery problem is thus the ability to recover a version of the complete system including all or part of this dynamic data and we approach this in several stages.

1. A complete description of the state of the system at an arbitrary instant involves specifying the state of every bit of storage, every processor, every peripheral and so on. In general it is virtually impossible to record such a system state on some external device since this would involve halting the entire system instantaneously and then extracting all the state information without perturbing it in any way.

2. If we wish to safeguard dynamic data against fault effects there seem to be only two basic ways round this problem, both of which involve redundancy during normal operation. The first is the conventional approach of setting up one or more complete extra systems to act as <u>standbys</u>, each of which does the same processing as the <u>on-line</u> system, so that they all contain the same or similar internal states (depending on how closely they are synchronised). The standbys thus act as complete and continuous records of the state of the on-line system. However, this involves a lot of redundancy which is not necessarily used very effectively, and as mentioned earlier, we are interested in the alternative, more general approach. This is to safeguard the on-line system by providing one or more extra storage devices, eg discs, on which we record only a limited amount of state information, just sufficient to reconstruct a viable system from which processing using dynamic data can be re-initiated after a fault.

3. More specifically, after a major fault we must be able to drive the entire system (or some viable subset of it) into a state in which:-

   a. all the various data areas are self-consistent internally and mutually consistent with each other;

   b. the processing actions required to continue from that state are known, eg where each program should be entered;

   c. the data areas have a high probability of being "correct" ie consistent with the situation in the outside world and the past activities of the system.

A complete system state satisfying conditions a. and b. can be generated from the read-only code and constant data backstop alone. To wholly or partially satisfy c. requires that at least some of the read-write data areas must instead be recovered using dynamic data stored redundantly in the system during normal operation, but in such a way that a. and b. are still satisfied. This is not a trivial problem, but it can be eased considerably by noting that the overall state re-established using dynamic data and satisfying all these conditions is not necessarily one which the system actually passed through before the fault.

4. There are two main ways in which we can limit the amount of dynamic data which has to be safeguarded and made available for recovery:-

- limitations by type or extent: instead of trying to record the state of all read-write storage, safeguard certain selected data areas only;
- limitation in time: instead of trying to record data at every microstep, instruction step, etc, record it much less frequently at certain selected instants.

An obvious way to limit the types of data to be safeguarded is according to the penalties of losing each type in a fault. The database areas are the most important since they comprise the system's long term dynamic memory. Transaction data areas are next, because their loss requires current transactions to be reinput from the outside world. Local workspace areas are least important because they have no external significance and can in principle be regenerated by rerunning processes from suitable initial conditions (ie suitable database and transaction data). Similarly, to limit the frequency with which the various dynamic data areas are recorded, we can rank them according to how fast they change, and it is fortunate that the order is the same. That is, database areas change less frequently than transaction data, which in turn changes less often than workspace data, and the faster the data changes, the less worthwhile it is to try and safeguard it for recovery.

5. An example of limitation in time only is the "snapshot dump" used in batch processing systems. This involves stopping the complete system at infrequent intervals and recording its entire state. However, this is rarely feasible in real time systems because the snapshots need to be much more frequent, ie up to date, since it is not usually acceptable to rollback the outside world by several hours, say. Also, it is rarely acceptable to stop the system for long enough to record its entire state anyway.

Similar arguments apply to limitation by type only. It would be very difficult to maintain continuous records of selected data areas and even if it were possible, the chances of them being unaffected by a fault would be low because there would be very little 'decoupling in time'.

6. To be practicable, therefore, a selective redundancy approach requires a combination of these techniques, based on making redundant records of the state of selected dynamic data areas at selected points in time, in such a way that

there is an adequate probability that a viable system can always be reconstructed after a fault at any instant using the current dynamic records plus the read-only backstop. It is fairly obvious that the database areas must be safeguarded, and that temporary workspace areas are very difficult to safeguard. We will show later how transaction data can be dealt with as extensions of database data, but for the present let us consider how database data areas can be safeguarded. The key question here is how we select suitable times at which to update the redundant records of their states.

7. To answer this, we must recall that the systems we wish to protect are basically structured as networks of co-operating but asynchronous parallel processes, each executing a single thread functional program. Any selective redundancy scheme should therefore be compatible with this. In particular, it would seem undesirable that it should introduce a large amount of extra coupling or synchronism between otherwise asynchronous processes, for example by forcing all the processes in the system to halt simultaneously in some special state while the states of some of their dynamic database areas are recorded. Accordingly, the approach we have evolved in HIVE considers each asynchronous process and each database separately, and the epochs at which their states are to be recorded are defined locally and independently for each one, rather than system - wide or globally. Further, we find that suitable local epochs can be readily identified by exploiting the basically cyclic nature of these single thread processes as follows.

8. Consider a process P which executes some code C and accesses database area D. When it is run to process a given transaction it must be given access to the data T of that transaction, and to a local workspace area W which can be initialised appropriately from a read-only version held as part of the backstop. As it runs, it writes into W preparing updates for D and T. At the end of the execution these can be inserted in D and T, the access to T can then be lost, the contents of W can be discarded and the process enters an idle state waiting for the next transaction. The amount of dynamic information needed to specify the state of the process at any instant thus varies cyclically and is at a minimum when the process is idle between successive executions. The instant when a process enters such a minimum information state is called a regeneration point (by analogy with renewal theory) and this is the obvious point at which to update the redundant records of dynamic data areas associated with that process which are to be safeguarded, such as the database D.

9. Our basic model of the safeguarding and recovery procedures is thus the following. The system is regarded as a set of asynchronous functional processes and a set of separate read/write databases, where each database is a composite object consisting of 2 or more data areas on separate storage media. Each process is permanently bound to its particular code areas and may also be permanently bound to one or more of the databases. Each database is permanently associated with at least one process. Each process executes cyclically to process transaction data acquired in ways to be described later.

During the execution of a process P associated with a database D, D is locked against all other processes and P is given read-write access to one version of D only, the <u>working version</u>. We now require that if any faults or inconsistencies are detected during the execution of P they must either be corrected or else P must be prevented from completing its cycle. This means that if P does complete its execution ie reach its next regeneration point, we may assume that it has executed correctly as far as it can tell, and in particular that the working version of D now represents a new, self consistent state for D, as far as P can tell. The working version can thus be used to update the other version(s) of D one at a time, so that there is at the most only one version of D which is inconsistent at any time due to normal updating action. Similarly, if P does not reach its regeneration point because of a fault which cannot be corrected, the <u>minimum</u> action required is to backdate the working version from one of the others, so that for example P may be re-run. (An alternative, nearly equivalent scheme is to give P read-only access to a version of D and allow P to prepare updates for D in its own local workspace W. The various versions of D can then be updated from W one at a time if and when P completes its cycle successfully. If not, no action is needed on D except to unlock it).

If a crash occurs at any instant, all the processes in the system are re-established in their idle states and with each database in the state recorded at its last regeneration point before the crash. (Checksum or other facilities are used to ensure that a partially updated or corrupted version can be detected, and this must be updated or backdated from other versions). This therefore satisfies 2 of the conditions given earlier for a selective redundancy scheme, because the reconstituted databases are always self consistent (they always correspond to a regeneration point ie successful completion of a process's cycle) and for the same reason the processing action needed to continue from then on is known. The other conditions, ie consistency between different databases and with the outside world, both depend on how transaction data is handled and safeguarded and this will be discussed later in the description of HIVE.

Finally, we note that with this approach it is helpful for some purposes to regard the state of the overall system at any instant as being defined basically by the various read-only and read-write data areas which are safeguarded on backing store, rather than by the state of all the disc and core locations in the system. This <u>safeguarded</u> state is thus the state that would be established in the system if it crashed at that instant, and it changes in discrete steps each time a process reaches a regeneration point. Rather than safeguarding what is in core by recording snapshots or checkpoints at various times, this approach thus is based on allowing processes to make dynamic incursions into core to try and reach their next regeneration point. If this is successful, the safeguarded state is changed in a discrete step. If it is not, the safeguarded state remains the same.

8. ENFORCEMENT OF STRUCTURE AND PROTECTION

We noted earlier that whatever structural concepts and components have been used in designing a system, it will finally be implemented and run as a set of storage areas containing code and data, located on discs and/or in core. To allow high integrity operation, the way in which these are organised and accessed must satisfy 2 main requirements, as follows.

8.1 Separation and Isolation

The conceptual separation and independence between the various code and data areas implied and required by the structual concepts must be enforced at all stages of development and operation. For example, if the execution of a given code module is only supposed to modify 2 particular data areas, it should be prevented from accessing any other code and data in the system as a result of design errors or hardware failures. Likewise, if a programmer specifies how a given data area is to be initialised at system build time, we must ensure that this area is only initialised from that source, and that the initialisation is not applied to other data areas. Other examples include ensuring that code is not modified during execution, that data is not executed as code, that read-only data is not written to, and so on.

This all requires a suitable blend of many different hardware and software techniques probably at several levels of the system and throughout the whole process of development and run time operation. Examples include hardware base-limit-access registers, compilers and run time checking software, and the design of these facilities is also heavily involved with fault detection requirements.

A particular requirement for separation and isolation has already been noted, in safeguarding a read-write data area by maintaining several versions of it on physically separate hardware with an updating procedure which ensures that versions are updated sequentially and that partially updated or corrupted versions can be detected after a crash. It is important to note that the access path to each version must also be protected to the same extent, so that for example, loss of one disc device does not cause loss of the information needed to find and access files on other discs.

8.2 Abstraction, Naming and Secure Access

So far, we have highlighted the need to enforce separation between the various code and data areas, However, the whole basis of the "levels of abstraction" approach requires us to associate simpler components together to form "higher level" ones. For example, a process is an association of certain code and data areas of various types.

Since ultimately the whole run-time system is held as the contents of store areas on disc and in core, all the associative or relational information about how areas are related and grouped must itself be held as the contents of such store areas, unless it has disappeared entirely by run-time. This can raise difficulties in enforcing separation and isolation unless considerable control is exercised over the means by which one component is able to reference another.

Consider the simple example of a process P consisting of a code area C, a workspace area W and a database area D executing on a transaction data area T. Assume each of these areas is held as a separate, contiguous segment of core defined by base and limit values, so that each area can only be accessed at run time by loading the appropriate base-limit values into one of the hardware protection registers of a processor. How can the code segment C gain access to the database segment D? If C is allowed to supply base-limit values directly, eg by computing them in the processor's registers, then we have no protection at all since C could supply any binary pattern it chose and this would be interpreted as base-limit values by the hardware. This would also be true if the base-limit values were obtained from locations in any read-write segment to which C can get direct access, or if C could supply any form of global name for D which the system will translate unconditionally into D's base-limit values. Three necessary conditions for maintaining protection follow from this:-

i Loading the base-limit registers (or equivalent operations) must not be done by application code but by trusted "system code" or microprogram forming part of the virtual machine implementation itself.

ii The parameters required by this code (ie the base-limit values etc) must be specified by the applications code as location parameters, and the locations concerned must be in a segment which cannot be accessed directly by the applications code, only by system code. That is, applications code can only refer to a segment by quoting a name for the "system location" holding the access information.

iii The name used to refer to a segment must be local in some sense to the program, process or whatever is requesting the access. That is, it must be automatically or implicitly prefixed with the (global) identity of the program or process making the request, so that it is only looked up or interpreted in a system area associated exclusively with that program or process. If not, ie if segment names are global, not local, then a program could specify any global name it chose and gain access to the corresponding segment, if it exists, just as though it had specified the actual base-limit values.

(Note: From the protection viewpoint, the use of apparently global names plus a list of which processes, programs etc are allowed to access each "global" name is equivalent to the local name scheme since access still depends on the identity of the requesting process).

All these various requirements are incorporated in the HIVE capability mechanism outlined below. The basic result is a means whereby data areas can securely contain references to other data areas in terms of locally defined names. These local names are interpreted by system code at run time as specifying particular locations within system data areas which in turn specify the type, current physical location and current status (eg locked) of the area(s) to be referenced. Associations between various application areas can thus be represented securely within such areas as relations between the appropriate local names.

9. THE HIVE VIRTUAL MACHINE

We have discussed at some length a number of different aspects of structure - transactions, functions, code and data, physical representation, parallel processing and single thread programs, levels of abstractions and virtual machines, redundancy, regeneration points and the enforcement of separation. We now wish to show how the high level virtual machine approach can provide a framework for all these structural concepts, by giving a short description of the main points of the HIVE virtual machine at RSRE. More detailed descriptions of some aspects will be found in Refs 1-3.

The facilities and resources provided by the HIVE virtual machine are the high level equivalents of those of the hardware virtual machine VM(o) mentioned earlier, as follows, (see Fig 3).

9.1 Virtual Processors

A HIVE virtual machine provides an arbitrary number of <u>virtual processors</u> (VPs) which apparently run asynchronously and independently in parallel. They are analogous to the physical processors of the hardware in that they incorporate a basic path-of-execution logic and can be applied to execute processes, eg a functional program on some data. VPs are the basic entity to which other HIVE resources can be allocated, and VPs are multiprogrammed together over whatever real processors exist in the hardware. However, HIVE VPs have certain additional properties compared to hardware processors, in order to simplify the applications programmer's task. Apart from the fact that we may have an arbitrary number of VPs, the main difference is that each VP in a HIVE system is dedicated at system build time to execute a particular functional program, and does so for the data of only one transaction at a time. This means that such programs can be written in single thread form, and that a VP must be <u>non-interruptible</u> in the sense that the arrival of a request to process another transaction cannot affect the VP's processing of its current transaction (see below). Note that we may still of course allocate several VPs to perform the same function on different transactions in parallel, ie the mapping between functional programs and processes executing on VPs is not necessarily one-to-one.

9.2 Store Resources - Protected Objects

The basic HIVE store resources are a set of <u>protected objects</u> in core (<u>core segments</u>) and on disc (<u>file segments</u> or <u>files</u> for short) which correspond roughly with the store areas we have discussed hitherto. Protected objects are grouped into a number of logically separate <u>regions</u>, some of which must also be physically independent. Each protected object is defined by a <u>resource descriptor</u> which defines the object's physical location, type (eg code) and status (eg locked). All the resource descriptors for objects in a given region are themselves held within a predefined protected object in that region, so that the region can be logically self-contained. The implementation of a HIVE system must include a suitable set of hardware (eg base-limit-access registers etc) and software facilities (eg run-time checks on capabilities, file accesses etc) to ensure that physical separation between protected objects is enforced.

9.3 Capabilities

A VP is given access to the various protected objects comprising the process it is to execute, such as code and data segments in core and on disc, by being given capabilities for them, on a "need to know" basis. In the present implementation each VP is represented by a system data area called its VP Base which defines various parameters of the VP, (eg its priority), provides space for saving its current register image and state data, and also defines all the capabilities the VP currently has by means of a caplist. The size of each VP's caplist is defined at system build time, and each entry may be void or may contain a resource descriptor index ie a reference to a resource descriptor in the form of an index down a set of resource descriptors. It also contains a field specifying the kind of access that VP is allowed to the resource (eg read-only, read-write, execute-only etc).

A VP refers to a resource for which it has a capability (ie a non-void entry in a pre-declared slot in its own caplist) by quoting its local name for that resource, which in HIVE is ultimately just the index number of the appropriate slot in its caplist. By defining suitable macros, applications programmers may of course use arbitrary local names in their source code. Different VPs may thus share access to the same resource by each having a capability for it, specifying different access rights for each VP if required, and each VP will thus have its own local name for the resource. The set of capabilities currently in a VP's caplist defines its current protection regime (see Fig 2) and no VP is allowed to have capabilities for any system code or data areas such as VP Bases, resource descriptors, etc.

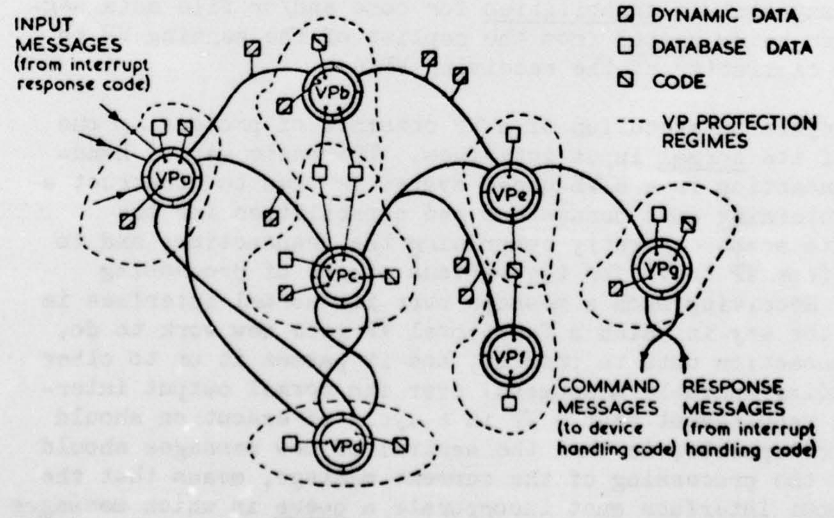

INPUT MESSAGES (from interrupt response code)
▨ DYNAMIC DATA
□ DATABASE DATA
⊠ CODE
--- VP PROTECTION REGIMES
VPa
VPb
VPc
VPd
VPe
VPf
VPg
COMMAND MESSAGES (to device handling code)
RESPONSE MESSAGES (from interrupt handling code)

Fig 2 Networks of HIVE VPs - Capabilities & Protection Regimes

A single capability may also be used to refer to a composite object consisting of a set of coexisting versions of a given data area, each held as a protected object in a different HIVE storage region. This is achieved simply by requiring that the resource descriptor index used to identify any version of a given object shall be the same for all regions so that a single capability holding a single resource descriptor index value is sufficient to refer to any object of the set. Which version will be accessed at any time then depends on which region(s), ie which resource descriptor lists, the capability is applied to and this in turn depends on the system code setting up the access. For example, normal run-time code might give access to either of 2 read-write versions of a database area, whereas recovery code might also be able to access 2 cold-start versions in the read-only backstop.

9.4 Message Passing

A VP's input/output facilities are provided by a message passing mechanism. Each VP has 2 message interfaces, the normal interface and the blocking interface (see Fig 2). Each interface consists of a set of input routes and output routes over which messages may be received and sent to other VPs in the system and also to hardware devices. Each VP again uses its own local naming scheme to refer to its own routes and so at run-time it is not aware of what its routes are connected to. Arbitrary fan-in and fan-out of routes is allowed, ie a given output route of a VP may be connected to input routes of several VPs and a given input route of a VP may receive messages from output routes of several VPs.

A message has 3 basic attributes:-

- it is a stimulus which may cause a VP to be scheduled to run;

- it may contain data parameters being passed directly from a data segment of the sending VP to data segment(s) of the receiving VP(s)

- it may contain capabilities for core and/or file data segments being passed from the caplist of the sending VP to the caplist(s) of the receiving VP(s).

The basic cycle of execution of a VP consists of processing one message off its normal input interface. The basic way of handling a transaction in a HIVE-based system is thus to construct a message containing data parameters and capabilities for the various data areas currently comprising the transaction, and to pass this from VP to VP for the various stages of processing required. Receiving such a message over its normal interface is therefore the way in which a functional VP gets new work to do, ie new transaction data to process, and it passes it on to other VPs by sending suitable message(s) over its normal output interface. The requirement that a VP in a cycle of execution should be non-interruptible, ie that the arrival of new messages should not affect the processing of the current message, means that the message input interface must incorporate a queue in which messages are buffered until the VP is ready to deal with them. This does not, of course, mean that the hardware processors are non-interruptible. Indeed the execution of a VP can be suspended at many points at the hardware level, but the VP is unaware of this. Similarly, although we may say the VP is non-reentrant with respect to messages, ie it can only process one at a time, we require the code of applications programs in a HIVE system to be reentrant so that, for example, several VPs could execute it in parallel, using different data areas.

A cycle of execution is always terminated by the VP's application code calling the END virtual instruction (see below) which indicates that its regeneration point has been reached. One of the parameters of END is an "OR" mask on all the VP's normal input routes which allows the application program to exercise dynamic control over the selection of the message to be processed on the next cycle. On END, the routes allowed in the mask are inspected sequentially and the first message found on an allowed route is selected for processing on the next cycle. If no suitable message exists, the VP becomes idle until a message arrives satisfying the condition.

To facilitate local recovery actions, the normal output interface of each VP is also buffered. That is, when a VP executes the SENDN virtual instruction at any point in its cycle to output a message over its normal interface, the message is not despatched at that time but is held in an output buffer. It is only released into the rest of the system if and when the VP calls END. Thus, if the VP aborts or wishes to repeat its cycle because a fault has been detected, the messages in the output buffer can be deleted without having had any effect on the rest of the system.

In the course of a cycle, a VP may call on another VP to perform some task for it by sending it a message via a route on its blocking output interface. On this interface, output messages are not buffered but are despatched immediately. Such a route will usually be connected (at system build time) to a normal input route of the called VP where the message will be queued if necessary until the VP performs a normal cycle of execution to process it. When it calls END, a message is sent back to the blocking input interface of the calling VP to indicate completion. Meanwhile, the calling VP may continue processing and may then await one or more replies from one or more called VPs by executing the BLOCK virtual instruction. This again requires an "OR" mask to be supplied as a parameter, but this time the mask applies to the input routes of the blocking interface. BLOCK causes the VP's execution to be suspended unless or until a message is present on one of the allowed blocking input routes, when it delivered and processing continues.

Communication between VPs and hardware devices is also carried out using the same basic message passing mechanisms, and further details of these topics will be found in Refs 1-3.

9.5 Instruction Set

The instruction set which a VP can execute may be thought of as consisting of:-

- the basic instruction set of the underlying, real hardware processor(s), (excluding certain priveleged facilities, depending on the implementation) for manipulating data within protected objects in a VP's current protection regime;

- a set of virtual instructions (VIs) for manipulating the resources of the HIVE machine as such.

Several VIs have already been mentioned in passing eg END and BLOCK. It is immaterial to the applications programmer how a VI is implemented - whether as macros, procedures, hardware microprogram or even special purpose processors. A VI may be thought of as a procedure which is executed as part of the VP's application code except that all VIs are executed in a separate protection regime from that of any VP. This is because VIs usually involve access to critical "system" data areas such as VP Bases and resource descriptors needed to implement the virtual machine itself.

The VIs required form a fairly compact set, and the main VIs are as follows:-

CREATE, DELETE	- for creating and deleting protected objects (core and file segments) dynamically at run time via capabilities declared at system build time
SEIZE, USE, RELEASE	- for accessing protected objects at run time in various ways, eg exclusive read-write access to shared core or file segment
SENDN SENDB	- for sending messages over the normal and blocking interfaces
END BLOCK	- for controlling message queues and the cycle of execution, including sequential updates of safeguarded databases
CALL, RETURN	- for transferring control between different code segments
MOVE	- for moving capabilities with a VP's caplist
READ, WRITE LOAD, FREE	- for moving data and between core and file segments
CHECKPOINT CREATE DISCAP RECOVER DISCAP DELETE DISCAP	- for creating and manipulating safeguarded copies of dynamically-created objects eg transaction data files (see below)

In the present implementation the VIs are provided as a set of Coral 66 recursive procedures which can be multithreaded, ie executed in parallel by a number of VPs. The VI protection regime is defined by a set of capabilities in a caplist, the public caplist, which is not part of any VP Base and which can only be accessed via a particular hardware instruction. VIs are invoked by inserting standard macro calls in the source code of an applications program and the macro expansions deal with parameter passing and the insertion of the call on the special hardware instruction. Some further discussion of the implementation eg the various levels involved, the ways in which VIs can call each other, and the use of a combination of VPs and VIs to implement the filing facilities themselves, will be found in the References.

9.6 Languages

As mentioned earlier, one of the logical consequences of the HIVE approach to system architecture is that the applications programmer requires two complementary types of formal language in order to implement a system. The first is a local programming language for specifying the sequential, single-thread algorithms for individual functional programs. More or less any conventional high-level programming language would be more or less suitable for this, providing that:-

a. it allows the insertion of calls on virtual instructions (eg as code macros) as well as real hardware instructions,

b. it allows the object code and data of a program to be organised as a number of separate, relocatable segments suitable for mapping into HIVE protected objects referred to via HIVE capabilities and accessed via hardware base-limit registers or the equivalent at runtime.

Obviously many other factors enter into the choice of a suitable language eg efficiency, block structure, data representation facilities, style, degree of protection and checking at compile and runtime, and not least, availability. The local programming language used in HIVE at present is Coral 66 (Refs 3 and 4) and HIVE is presently implemented on a Computer Technology Modular One system which provides only 3 base-limit registers in each processor for protected access to core. The Coral 66 compiler for this machine allows the object code and data it generates to be consolidated into a number (> 3) of separate segments but otherwise it is an implementation of standard Coral 66 and we have made no significant modifications in order for it to be used to generate code for HIVE VPs. Any of the normal programming constructs and structures allowed by the language - tables, arrays, procedures, blocks etc - can be used freely within a set of protected objects constituting a runtime process executing on a VP.

The second type of language required is a system description language. This is quite a different sort of language and its purpose is to enable system designers and application programmers to specify how individual programs written in the local, algorithmic language(s) are to be mapped on to a set of HIVE protected objects and VPs, and how these VPs are to interconnected to form a complete, executable system. Specifically, the system description language SYDEL which we have defined and implemented for HIVE has 3 main aims:-

a. to allow a complete system to be described in a way which is sufficiently formal and precise to be held as a database from which a suitable "back end" system building program can automatically build the initial core and disc images of a complete executable system (including all the correctly initialised system data areas such as VP Bases) given the object code and data segments of its applications programs;

b. to allow the SYDEL database representing the desired system to be built up incrementally over an extended period of time as information about its various components becomes available;

c. to design SYDEL itself as a general, extensible language which is not specific to HIVE by providing facilities in the language allowing the components of the system to be described (eg VPs, Coral programs etc) to be defined dynamically as part of the SYDEL database itself, rather than as part of the SYDEL compiler.

SYDEL is not, therefore, an algorithmic language for specifying sequential programs, but rather an assertive language for making statements about the structure of a system and the relations between its components, eg between programs and VPs, VPs and VPs and so on. It is currently implemented in Coral 66 on our Modular One machine and further details are given in Ref 5.

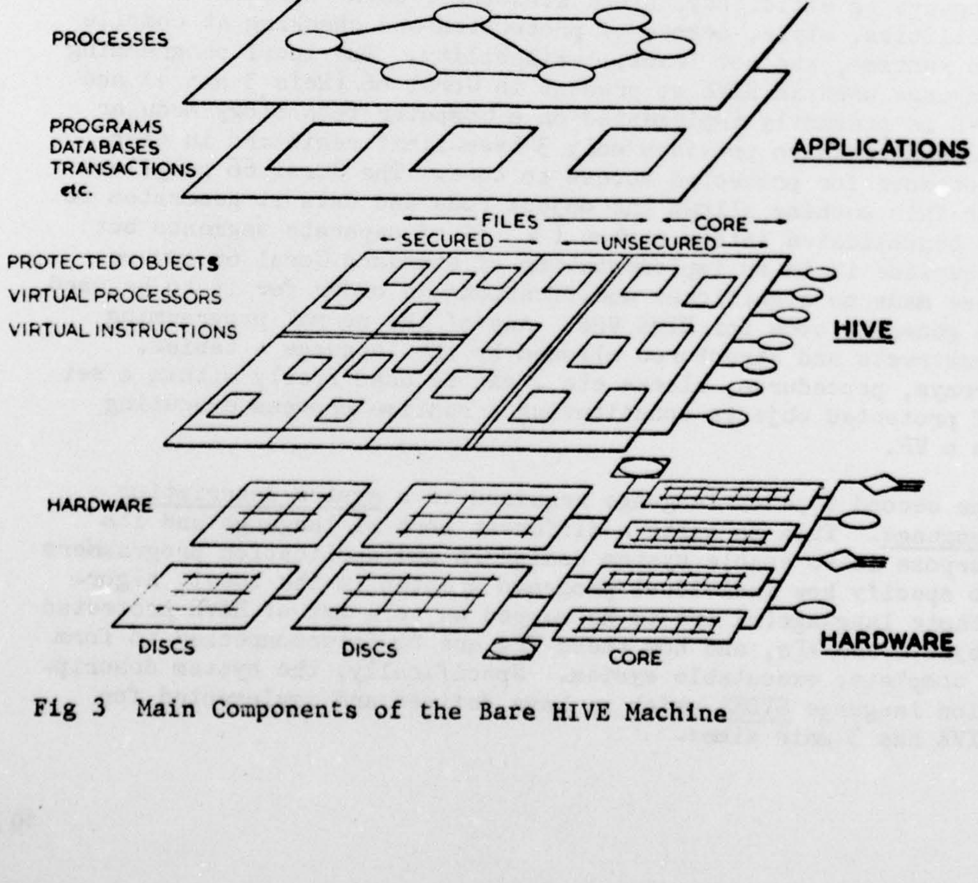


Fig 3 Main Components of the Bare HIVE Machine

10. MAIN POINTS OF HIVE

HIVE as described so far may be summarised as follows:-

The "bare HIVE machine" (Fig 3) provides a set of asynchronous, independent virtual processors, a set of protected objects (core and file) existing in various regions, and an instruction set consisting of the instructions of the basic hardware plus a set of virtual instructions for manipulating HIVE resources eg protected objects and messages, as such. A VP may access a protected object only if it has a capability for it in a slot in the caplist in its VP Base.

VPs can communicate with each other and the outside world by sending and receiving messages over the input and output routes comprising their normal and blocking interfaces, where messages are queued in various ways if necessary. In a running, HIVE-based system, each VP is assigned permanently to execute a particular functional applications program in a cyclic manner, processing one message at a time. Capabilities for the necessary code and permanent database areas can therefore be given to VPs permanently at system build time and the connectivity of routes between VPs can also be set up permanently at this time. The individual applications programs are single thread and written in a local, conventional high level language (currently Coral 66). A system description language, SYDEL, is provided to describe the mapping between programs and VPs and the interconnection between VPs. The SYDEL compiler builds up a database containing this information which can be used to build the initial care and file images of the initial run time system.

A VP may create new protected objects dynamically at runtime, referenced via capabilities which it must have declared at system-build time, and it may pass such objects securely to other VPs by sending a message containing the appropriate capabilities. When a VP receives such a dynamically created object in a message from another VP, a capability for it is copies from the message into a predeclared slot in its caplist, via which it may then be referenced.

Thus a VP may create a new data object, referencing it via a predeclared and previously void capability in its caplist, say C5. It can then load it with data, send it to other VPs in a message, delete its own capability for it, and repeat this sequence, if required, during its current cycle of execution. It can use the same local name (C5) for each new object and the same set of Coral 66 declarations within each new object, and it has secure, protected access to each new object until it has finished with it. HIVE thus allows the binding between code and data objects to change dynamically but securely at runtime, through the intermediaries of VPs, messages and capabilities, whilst still retaining the use of conventional high level languages such as Coral 66.

A running HIVE-based system thus consists of a network of interconnected, functional VPs (Fig 2) with transactions moving through this network as data and groups of capabilities in messages. Such a system must be implemented at the HIVE virtual machine level as a set protected objects, since this is all that is available, which will in turn be mapped onto actual hardware resources at lower levels. Higher level abstractions such as programs, processes (to run on VPs), transactions, databases, subsystems of processes, etc are represented at runtime by allowing capabilities for protected objects to be held in VP Bases and messages, and by then allowing VPs to form arbitrary relationships between the capabilities it has itself, by manipulating its own local names for them within its own protected objects.

11. REDUNDANCY AND RECOVERY IN HIVE

We may now describe the facilities provided in HIVE to support the implementation of the selective approaches to data safeguarding and recovery outlined earlier. Following the 'outside in' approach as before, we begin by considering various stages or levels of major recovery using progressively more dynamic data, and then consider stages of progressively more minor or local recovery involving less disruption to the overall system.

11.1 Cold Start Global Recovery

This is achieved by providing multiple copies of the read-only backstop, which consists essentially of the initial core and disc images constructed by the system building facilities from the representation of the system in the SYDEL database. The initial core image contains all the virtual instruction code and initialised versions of all the 'system' data areas such as VP Bases and resource descriptors. Loading this into core therefore corresponds to re-establishing the network of applications VPs with all their permanent capabilities for code and database objects and routes, with the database objects initialised to their "cold Start" states. This "empty" network of VPs initially contains no messages or dynamically created data objects.

11.2 Database Recovery

Safeguarding and recovery of the contents of read-write file objects is achieved by means of the following procedures and facilities during normal operation. Each database object is declared as a composite object comprising at least four versions, namely two read-only, cold start versions forming part of the backstop, and two read-write versions for use at run-time. When a VP V having a permanent capability for such a database D begins a cycle of execution, D is locked out to other VPs and V is given access to only one of the read-write versions of D, the working version. If and when V completes its cycle successfully and reaches its next regeneration point, the END virtual instruction then causes the other read-write version(s) of D to be updated one at a time from the working version. If the regeneration point is not reached, the working version is backdated from one of the others. When updating or backdating are complete, D is unlocked for further access by any VP with a suitable capability.

To reduce overheads, the filing system code keeps track of which pages of the working version have actually been modified during the current cycle, and only these pages are updated in the other version(s) by END. For each version of a file, a checksum is also maintained in the file header for each page of the file. These checksums are only updated in the header when all updating of the file pages in that version is complete. This enables partially updated or corrupted files to be detected during recovery.

If a major system crash occurs at some arbitrary instant, the various discs in the system will contain:-

a. the various replicated read-only files comprising the backstop;

b. two or more versions of each of the permanently existing read-write database files, some of which may be internally inconsistent because they were in the course of being written to or because they were corrupted in the crash;

c. various dynamically created files which are not replicated, holding transaction data etc.

The basic principle used in recovering the discs after a crash is that the only file objects reconstituted are those accessible via capabilities in the initial core image of the read-only backstop. Basically, therefore, the only file objects which can be accessed after a major crash are those for which permanent capabilities have been inserted in a VP Base at system-build time. Two ways in which this is extended to allow safeguarding of transaction data are described later.

The recovery procedure for database files is thus as follows. First, the read-only initial core image is bootstrapped in from the backstop, as for a cold start, and this image also contains recovery code, which is entered. Its main task is to garbage-collect the filing system by going through all the VP Bases in the initial core image and examining in turn each permanent file capability in each caplist. For each such capability, it tries to access each version of the file in turn. For each version which is still accessible, the checksums are evaluated to detect partially updated or corrupted pages, and where possible the appropriate updating or backdating from other versions (including the cold start versions if necessary) is carried out. When this process is complete, any areas of discs which have not been accessed are placed on the free space list of the filing system and full normal processing can then continue. The basic effect of this level of recovery is thus to re-establish an 'empty' network of idle applications VPs together with their permanent read-write database files, with each such file in the state prevailing at the most recent regeneration point of the last VP to access it.

11.3 Transaction Recovery

At least two approaches are possible to the safeguarding of transaction data which is normally to be processed in the system as dynamically created and unduplicated core or file objects. They are both based on forming <u>checkpoints</u> of such data at suitable stages in its processing (eg as soon as it has first been input to the system and identified as a transaction). Different approaches result from requiring different degrees of isolation and protection on the checkpoint data of different transactions.

The first approach is simply to form transaction checkpoints by writing the data of each transaction into a common, permanent, safeguarded database file (called a checkpoint file) which can be accessed and recovered in the way already described after a crash. Any VP which has a responsibility for forming checkpoints is designated a checkpoint VP although it may have other functions as well. When a crash occurs, the various transactions then in the system could have been at any stage of processing and many indeed have been on the input queues of several VPs at once. The recovery actions required are thus to reconstitute the "empty" network of VPs and their associated read-write databases, as described already, and then to run the checkpoint VP(s). For each transaction currently represented in its checkpoint file, each checkpoint VP can then create suitable dynamic core or file objects and initialise them from the data in the checkpoint file, which still remains as a safeguarded database file in case of further crashes. The checkpoint VP may then initiate the reprocessing of the transaction by sending a message containing capabilities for the newly-created transaction data objects to the appropriate VP(s) in the reconstituted network.

Before the crash, a given transaction T may have been processed by some VPs in the network but may not have reached others. If T is inserted for reprocessing from a checkpoint after a crash, it will therefore in general pass through some stages of processing twice (or more if a series of crashes occurs). This may be reflected in the contents of safeguarded databases associated with these VPs, and also possibly in signals sent to the outside world. Ensuring that a transaction may in fact be processed satisfactorily by a VP which has already processed it before is an application - dependent problem. It is equivalent to satisfying two of the recovery conditions given earlier, ie that databases are mutually consistent with each other, with transaction data and with the past history of the system.

HIVE provides a basic facility to assist in this, in the form of a marker bit in every message. When set (by a checkpoint VP) this indicates to any receiving VP that the message originates from the initiation of a reprocessing attempt on the associated transaction. Once set, the bit remains set in all subsequent messages generated in the course of that reprocessing attempt, and so each VP can determine whether it may have seen that transaction before. The general problem of defining the principles to be used in designing a HIVE application system containing a set of asynchronously - safeguarded databases so that transaction recovery is always possible could be highly relevant to the design of physically distributed systems, where this sort of approach may be unavoidable.

The second approach to safeguarding transaction data is in fact just a more sophisticated version of the first, providing greater isolation between the checkpoint data of different transactions.

This is done by checkpointing the data of each transaction in a separate dynamically created file which is itself safeguarded. This in turn involves:

a. allowing the dynamic creation of composite file objects comprising 2 or more versions,

b. putting the capabilities for such files in special database files which provide safeguarded extensions on disc of a checkpoint VP's normal caplist in core. Suitable virtual instructions are needed to do this to maintain security.

In recovery, these disc-cap files are first accessed and checked out as database files by the garbage collector in the normal way. Then each dynamic capability in each such file is examined in turn and checked out in the same way, as though it were for a permanent database file. Transactions can then be reformed and reinserted into the reconstituted network of VPs in the way already described.

11.4 Local Recovery

Two less drastic forms of recovery are immediately possible with the facilities described above. First, individual transactions can be selectively reprocessed with very little perturbation to the system, by simply causing a checkpoint VP to re-insert suitable messages by reference to its checkpoint file or disc-cap file. Secondly, if an individual VP detects a fault which prevents it completing its current cycle correctly, it can attempt to repeat the cycle without affecting the rest of the system. This is because, if a repeat is required:-

a. any messages sent over the normal output interface are buffered until END and can be deleted;

b. any alterations to the working version(s) of any database(s) accessed by the VP can be reversed by back-dating from the other version(s).

12. MAIN POINTS OF HIVE REDUNDANCY AND RECOVERY

HIVE thus provides very considerable scope for implementing selective approaches to redundancy and recovery via a small number of quite simple concepts and facilities, which may be summarised as follows. The long-term read-write application data areas are organised as a number of separate databases each associated permanently with one or more VPs. Each database is held as a composite file object consisting of 2 or more versions on physically separate regions of HIVE storage, all referenced via the same capability. Database files are updated asynchronously whenever the VP(s) which access then reach a regeneration point, ie successfully complete a cycle by executing END.

Recovery after a major crash re-establishes an empty network of VPs with their permanent capabilities for code and database objects and with each database object in the state corresponding to the most recent regeneration point of its associated VP(s). Any dynamically created file or core objects for which capabilities do not exist in the read-only backstop core image will be lost on major recovery unless special precautions are taken using safeguarded disc-cap files.

The simplest way of safeguarding transaction data is to write checkpoints of it into safeguarded database files, from which reprocessing may be initiated following database recovery after a crash, by regenerating suitable dynamic data objects and messages which are sent to the appropriate VPs in the system. There are many different ways in which these facilities can be used in the applications software. For example, a VP with access to database D can also be given access to a second database D' which it only writes to once every 20 executions, say, in order to form a complete second line of defence for D. A number of less drastic, local recovery actions is also readily possible, including reprocessing selected transactions and repeating particular VPs without disrupting the rest of the system.

13. CONCLUSION

Dedicated systems present many different requirements - flexibility, high integrity, ease of development, efficiency, real-time response. Many different structural concepts exist, in terms of which various aspects of these systems can be described and designed, including transactions, functions, single thread programs and data, programming languages, levels of abstraction, co-operating parallel processes, selective data redundancy, isolation and protection, and various approaches to recovery. The HIVE high level virtual machine is based on a relatively small set of basic concepts and components - virtual processors, protected objects, capabilities, messages and routes, virtual instructions, conventional high level languages and a system description language. We believe it to be a significant step towards providing a more coherent, unified framework in which all the various requirements and aspects of dedicated, high integrity systems can be expressed.

14. ACKNOWLEDGEMENTS

HIVE has been the work of a number of people at RSRE including Mike Partridge, Howard Nichols, Fred Bell, John Dallenger, Keith Parks and Bruce Ogilvy-Morris.

15. REFERENCES

1. Taylor J M, Partridge M F, Nichols H K and Hill J S, "HIVE - A High Integrity Virtual Machine for Complex Dedicated Applications", Proc IEE Conf "Software Engineering for Telecommunications Switching Systems", Univ of Essex, April 73, IEE Conf Pub No 97.

2. Taylor J M & Nichols H K, "Coral 66 in the HIVE Virtual Machine", ibid.

3. Taylor J M, "Design of a Virtual Machine for Dedicated Applications", Infotech State of the Art Report on Computer Design, 1974.

4. "Official Definition of Coral 66", HMSO, May 1970.

5. Partridge M F, "SYDEL - A System Definition Language" SRDE Report - in preparation.

6. Taylor J M, "Structural Components for Dedicated High Integrity Systems", Infotech State of the Art Report on 'Real Time Software', 1976.

DOCUMENT CONTROL SHEET

(Notes on completion overleaf)

Overall security classification of sheet ....UNCL....

(As far as possible this sheet should contain only unclassified information. If is is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R),(C) or (S)).

1. DRIC Reference (if known)	2. Originator's Reference 76010	3. Agency Reference	4. Report Security Classification UNLIMITED
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location RRE CHRISTCHURCH DORSET UK		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title REDUNDANCY AND RECOVERY IN THE HIVE VIRTUAL MACHINE

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers). Title, place and date of conference

8. Author 1. Surname, initials	9a Author 2	9b Authors 3, 4...	10. Date	pp	ref
TAYLOR J N	-	-	5.1976	26	6
11. Contract Number	12. Period	13. Project	14. Other References		
-	-	-	-		

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

RECOVERY, REDUNDANCY, VIRTUAL MACHINE, FAULT TOLERANCE, HIGH LEVEL LANGUAGE, SYSTEM DESCRIPTION LANGUAGE, MULTIPROCESSOR, HIGH INTEGRITY, SOFTWARE STRUCTURE, DEDICATED SYSTEM, REAL-TIME, SYSTEM ARCHITECTURE

continue on separate piece of paper if necessary

Abstract

The HIVE project is concerned with studying high level virtual machine architectures suitable for designing and implementing large, high-integrity transaction processing applications such as communication switching and database access systems. The main aim of the work is to develop a unified set of structural concepts and components in terms of which all the different and often conflicting design aspects of such systems can be coherently expressed. The approach followed has been to embody these ideas in the specification of a high integrity virtual machine, HIVE, and to implement HIVE and evaluate it experimentally by using it to implement test-bed applications systems.

This work has involved considering not only the architecture of the overall run-time software during normal operation, but also various selective approaches to data protection, redundancy and recovery, and methods for making the task of compiling, generating and integrating all the software of an application in an incremental, evolutionary way.

This report is concerned primarily with the first two of these areas. It presents the main points of the HIVE system as it stands at present, with particular emphasis on the selective data redundancy and recovery aspects, and also discusses the rationale behind the main design principles of the system.